Tugas 1 Teori Komputasi

Carles Octavianus, 2006568613

Disclaimer: I understand there might be a better way to parse a grammar, especially in the context of a compiler. Right now, I don't know that method, so I'll try a more straightforward approach."

First Prototype: A Naive, Yet DFS-Based Approach

It's a straightforward problem; it involves finding paths in an infinite tree traversal that lead to terminal symbols, considering all possible productions/branches, and subsequently adding them to the language set

```
language = set()
get_atmost = 5
def construct_grammar(
    terminal_symbols: list,
    non_terminal_symbols: list,
    start_symbol: str,
    productions: list[tuple],
    solutions="",
):
    global language, get_atmost

    if len(language) >= get_atmost:
        return None

    if solutions == "":
        solutions = start_symbol

    if check_terminal(solutions, non_terminal_symbols=non_terminal_symbols):
        language.add(solutions)
        return None

    for alpha, beta in productions:
        if alpha in solutions:
            construct_grammar(
                terminal_symbols,
                non_terminal_symbols,
                start_symbol,
                productions,
                solutions.replace(alpha, beta, 1),
            )
```

Function to Check if the Solution Consists Only of Terminal Symbols or Not.

```
def check_terminal(solutions, non_terminal_symbols):
    for i in solutions:
        if i in non_terminal_symbols:
            return False
    return True
```

So, everything is good. Let's give it a try.

```
    terminal_symbols, non_terminals_symbols, start_symbol, productions = test_case_1()

    construct_grammar(
        terminal_symbols=terminal_symbols,
        non_terminal_symbols=non_terminals_symbols,
        start_symbol=start_symbol,
        productions=productions,
    )
    print(language)
```

```
RecursionError: maximum recursion depth exceeded in comparison
```

it doesn't work. Can you guess why?

```
    for alpha, beta in productions:
        if alpha in solutions:
            construct_grammar(
                terminal_symbols,
                non_terminal_symbols,
                start_symbol,
                productions,
                solutions.replace(alpha, beta, 1),
            )
```

This code implicitly assigns a rank to each element in the production by order. Consequently, it will always be checked first, second, and so on, and it will not converge if I use these ordered productions, for example:

```
productions = [("S", "aAa"), ("A", "aAa"), ("A", "b")]
```

Now, let's introduce an element of randomness when choosing productions.

Second Prototype: Adding Randomness Doesn't Solve the Issue

```
...

for i in range(3*len(productions)):
        alpha, beta = random.choice(productions)
        if alpha in solutions:
            randomize_construct_grammar(
                terminal_symbols,
                non_terminal_symbols,
                start_symbol,
                productions,
                solutions.replace(alpha, beta, 1),
            )

...
```

```
test_cases_1:{'aba', 'aaaaaaaaabaaaaaaaaa', 'aaaabaaaa', 'aaaaaabaaaaaa', 'aaaaaaaaaabaaaaaaaaa'}
test_cases_2:{'abaaaa', 'abaaaaaaaa', 'abaaaaaa', 'abaaa', 'abaa'}
```

It's solved for test_cases_1 and test_cases_2, but it doesn't even converge for test_cases_3. Can you guess why?

Also, in test_cases_2, it doesn't seem right; it will result in a solution in the form of `ab^n`, which is not what we want.

As we know, the solution for test_cases_3 is in the form of $\{a^n b^n c^n | n \geq 0\}$. It's definitely worth attempting to find the solution systematically. Just imagine if we're in the state `aaabX`, the chances of transitioning to `aaabbbccc` are certainly slim. Therefore, we need to approach it in a more systematic way.

Third Prototype: A Greedy Approach to Assess the Rank for Each Element in Productions

Not all productions (mappings) are created equal. Some bring us closer to the solution, while others lead us further away from it. Therefore, it's essential to rank each element in the productions. Several methods can accomplish this, but I will employ a straightforward greedy criterion to rank each mapping in the productions:

1. Mappings that don't add anything, such as ("BC", "CB"), belong to the lowest tier and are checked last.
2. Mappings that reduce the result to a terminal symbol, like ("A", "a"), are more favorable and are checked first.
3. Mappings that introduce additional terminal symbols, such as ("A", "aAa"), are ranked second.
4. Initial mappings that incorporate non-terminal symbols, such as ("S", "aAa"), are ranked third.

Is this the optimal solution? I don't believe so, but it's sufficient for test_cases_3.

Therefore, with this approach, we will modify the productions from:

```
    productions = [
        ("S", "abC"),
        ("S", "aSBC"),
        ("CB", "BC"),
        ("bB", "bb"),
        ("bC", "bc"),
        ("cC", "cc"),
    ]
```

to:

```
    productions = [
        ("bB", "bb"),
        ("bC", "bc"),
        ("cC", "cc"),
        ("S", "abC"),
        ("S", "aSBC"),
        ("CB", "BC"),
    ]
```

We have a `reorder_productions` function that reorders the productions based on the criteria mentioned above.

```
def reorder_productions(
    productions,
    terminal_symbols,
    non_terminal_symbols,
):
    tier_1_productions = []
    tier_2_productions = []
    tier_3_productions = []
    tier_4_productions = []
    new_productions = []

    for alpha, beta in productions:
        beta_counter = [beta.count(i) for i in terminal_symbols]
        alpha_counter = [alpha.count(i) for i in terminal_symbols]

        if check_terminal(beta, non_terminal_symbols):
            tier_1_productions.append((alpha, beta))
        elif sum(beta_counter) > sum(alpha_counter) and alpha != "S":
            tier_2_productions.append((alpha, beta))
        elif alpha == "S":
            tier_3_productions.append((alpha, beta))
        else:
            tier_4_productions.append((alpha, beta))

    new_productions.append(tier_1_productions)
    new_productions.append(tier_2_productions)
    new_productions.append(tier_3_productions)
    new_productions.append(tier_4_productions)
```

We also introduce randomness within each tier when selecting the productions.

```
{...}
    for tier in productions:
        for i in range(3 * len(tier)):
            alpha, beta = random.choice(tier)
            if alpha in solutions:
                construct_grammar(
                    terminal_symbols,
                    non_terminal_symbols,
                    start_symbol,
                    productions,
                    solutions.replace(alpha, beta, 1),
                )
{...}
```

It solves test_cases_3, but for larger values of n, it takes a significant amount of time to converge. I'll leave it as it is for now, but I plan to work on improvements in the future.

Let's revisit test_cases_2; it's still not correct, as it still produces strings in the form of `ab^n`.

Fourth Prototype: Switching from DFS to BFS

As we observe, the primary issue with the previous prototypes (from the first to the third) is that they follow a depth-first approach instead of a breadth-first one. Consequently, other branches of the solution may not even be discovered.

```
def construct_language_bfs(
    terminal_symbols,
    non_terminal_symbols,
    start_symbol,
    productions,
    solutions="",
):
    language = set()
    queue = [solutions]
    while queue:
        solutions = queue.pop(0)

        if len(language) >= get_atmost:
            return language

        if solutions == "":
            solutions = start_symbol

        if check_terminal(solutions, non_terminal_symbols=non_terminal_symbols):
            language.add(solutions)

        for alpha, beta in productions:
            if alpha in solutions:
                queue.append(solutions.replace(alpha, beta, 1))
```

It solves test_cases_1 and test_cases_2, but it also faces challenges when converging for test_cases_3 due to the specificity of the terminal nodes.

Thank You